National Defence / Défense nationale

A SIMULATION SYSTEM BASED ON THE ACTOR PARADIGM

by

J. McAffer
Radar ESM Section
Electronic Warfare Division

DEFENCE RESEARCH ESTABLISHMENT OTTAWA
TECHNICAL NOTE 89-4

PCN
041LB14

February 1988
Ottawa

89-04624/

ABSTRACT

The main goal of this work is to produce a sophisticated object-oriented design tool for simulating real-time applications. The basis of this tool is Actor-based simulation, a powerful mechanism for modelling real world objects simply, quickly and intuitively. It combines features of event-based and clock-based simulation and allows designers to use both in a common framework. Actors can be used to represent everything from the most basic to the most complex components. Complex objects are constructed by modelling their subcomponents and interactions. The Actor paradigm provides a good model of concurrency and facilitates the development of comprehensive monitoring and debugging tools.

RESUME

L'objectif de ce travail est la production d'un outil de conception complexe basé sur une structure d'objets pour simuler des applications survenant en temps-réel. Le principe de cet outil est la simulation d'Acteurs, un puissant mécanisme pour modeler des objets réels simplement, rapidement et intuitivement. Cette simulation combine les particularités de la simulation d'évènements et de la simulation de temps et permet au modeleur d'incorporer les deux dans un même cadre. Les Acteurs peuvent tout représenter, du composant le plus simple au plus compliqué. Les objets compliqués sont bâtis en modelant leurs sour-composants et leurs interactions. Le paradigme de l'Acteur offre un bon modéle de simultanéité et facilite le developpement d'outils de contrôle et de dépistage d'erreurs.

EXECUTIVE SUMMARY

DREO is developing sophisticated object-oriented software tools for designing, simulating and developing real-time applications. Designers need software tools which allow them to prototype and test models of complex real world objects. Since implementing a prototype that runs in real-time is often difficult and expensive, prototypes are implemented in simulated real-time. Simulation gives the prototype designer control over the time dimension of the system and allows him to test the system in a controlled environment. The effects of introducing new algorithms or features are determined quickly and easily, making simulation both cost and time effective.

Typical simulation systems are difficult and very time consuming to use. Validation of simulation strategies is also difficult because the modelling code is large, unreadable and littered with special simulation related commands. Modelling of concurrent systems is even more difficult to implement and validate. We have developed a simulation system which is based on object-oriented programming (OOP) principles and results in understandable simulations. Much less programming time is required as simulation designers can model systems in a more inituitive fashion. Because the resultant simulations are simpler and easier to understand, they are easier to validate. The Actor concept is used to describe the concurrent components in the simulation. Actors provide communication and coordination facilities which are transparent to the simulation designer. These features combine to create a simulation system which allows designers to concentrate on modelling real world objects rather than on programming.

The concept of time within a simulation can be implemented in a number of different ways including: TimeWarp, TimeLock and Clock Hierarchies. These mechanisms have been considered and the Clock Hierarchies has been chosen for use in the implementation of an Actor-based simulation system. A description of the implementation of this system is presented including the message passing coordination mechanism. The implementation was tested by creating an emulation of the Harmony real-timing operating system.

As a result of this work it has been found that Actor-based simulation is a very real possibility and thus a very useful tool in the design and development of future real-time systems. This will result in time and cost savings and will result in higher quality systems.

TABLE OF CONTENTS

		PAGE
ABSTRACT/RESUME.		iii
EXECUTIVE SUMMARY.		v
TABLE OF CONTENTS.		vii
LIST OF FIGURES.		ix
1.0	INTRODUCTION	1
2.0	ACTORS	1
2.1	Example Actors	2
2.2	Actor Communication.	2
2.3	Software Development Features of Actors.	5
2.4	Implementation of Actors in Smalltalk.	5
2.4.1	Classes Required for Actors	5
2.4.1.1	Actor.	6
2.4.1.2	ActorMessage	6
2.4.1.3	Encapsulator	6
2.4.2	How Actors Pass Messages.	10
2.4.3	Monitoring.	11
3.0	ACTOR-BASED SIMULATION	11
3.1	Time, Events and Actors.	12
3.2	The Clock hierarchy.	14
3.3	Implementation	14
3.3.1	Required Classes.	14
3.3.1.1	SimulationActor.	14
3.3.1.2	Clock.	15
3.3.1.3	SimulationObject	15
3.3.1.4	SimulationSystem	15
3.3.1.5	SimulationMessage.	15
3.3.2	SimulationMonitor	16
4.0	HARMONY SIMULATION/EMULATION	16
4.1	An Implementation of Harmony Emulation	17
4.1.1	HarmonyTask	17
4.1.2	HarmonyDirectory.	17
4.1.3	HarmonyProcessor.	17
4.1.4	HarmonySystem	17
4.2	Results.	17
5.0	CONCLUSIONS.	18
6.0	REFERENCES	19
7.0	ACKNOWLEDGEMENTS	19

LIST OF FIGURES

		PAGE
FIGURE 2.1:	TIME LINE OF INTERPROCESS COMMUNICATIONS IN HARMONY	4
FIGURE 2.2:	OBJECT POINTERS BEFORE AND AFTER A TWO WAY become: OPERATION	7
FIGURE 2.3:	SIMPLE MESSAGE MONITORING TECHNIQUE	8
FIGURE 2.4:	MONITORING ESSAGE TRAFFIC USING doesNotUnderstand:.	9
FIGURE 2.5:	PSEUDO-CODE FOR AN ACTOR ENCAPSULATOR'S doesNotUnderstand: METHOD.	10
FIGURE 3.1:	SIMPLE MODEL OF A REAL WORLD OBJECT	12
FIGURE 3.2:	COMPLEX MODEL OF A REAL WORLD OBJECT.	12

1.0 INTRODUCTION

DREO is developing sophisticated object-oriented software tools for designing, simulating and developing real-time applications. Designers need software tools which allow them to prototype and test models of complex real world objects. Since implementing a prototype that runs in real-time is often difficult and expensive, prototypes are implemented in simulated real-time. Simulation gives the prototype designer control over the time dimension of the system and allows him to test the system in a controlled environment. The effects of introducing new algorithms or features are determined quickly and easily, making simulation both cost and time effective.

Typical simulation systems are difficult and very time consuming to use. Validation of simulation strategies is also difficult because the modelling code is large, unreadable and littered with special simulation related commands. Modelling of concurrent systems is even more difficult to implement and validate. We have developed a simulation system which is based on object-oriented programming (OOP) principles and results in understandable simulations. Much less programming time is required as simulation designers can model systems in a more intuitive fashion. Because the resultant simulations are simpler and easier to understand, they are easier to validate. The Actor concept (Ref. [6]) is used to describe the concurrent components in the simulation. Actors provide communication and coordination facilities which are transparent to the simulation designer. These features combine to create a simulation system which allows designers to concentrate on modelling real world objects rather than on programming.

In this report we detail how an object-oriented programming (OOP) environment is extended to support the Actor concept and how this concept is used to create a sophisticated simulation environment. Section 2 describes a novel approach to the implementation of Actors as an extension of Smalltalk as well as some of the benefits of using Actors and an OOP environment. Section 3 introduces the idea of Actor-based simulation, a very powerful and straightforward concept upon which simulations of simple and complex real-time applications can be based. One such simulation system is discussed and its implementation described. A further section gives an example of Actors and Actor-based simulation in the form of an emulation of Harmony (Ref. [4]), a real-time operating system, running on a multiprocessor system.

2.0 ACTORS

A real-time system simulation environment must provide an acceptable model of concurrency. The Actor concept is one such model. Originally Actors were presented as conglomerates of objects which were independent and functioned asynchronously with respect to the other conglomerates in the system. It is difficult to provide a more specific definition which would be widely accepted since particular implementations have tended to deviate substantially from the original concept. Saying that Actors are active objects summarizes our definition. Actors are encapsulations of data, operations on this data, communications capabilities to allow these operations to be used and the computing power required to drive all of the above. In other words, an Actor behaves like a Smalltalk object which has a process and communication mechanism attached to it.

Actors provide a clean package with which simulation designers can model concurrency as well as functionality. The simplicity and functionality of Actors allows designers to combine single Actors and groups of Actors into meaningful entities which function concurrently.

Smalltalk was chosen as the implementation environment for an Actor-based simulation system because its objects already have many of the properties desired for Actors and Smalltalk provides sophisticated development and debugging tools. Unfortunately, Smalltalk's model of concurrency is limited to Smalltalk processes running under the control of the Smalltalk interpreter which is itself a single operating system process. Further, Smalltalk processes are not "first class" objects, but rather must be treated specially using a limited set of operations. Both Smalltalk-80 (Ref. [9]) (Ref. [5]) and Smalltalk/V (Ref. [3]) are subject to this limitation. Since an Actor's methods and protocol implicitly define its related process' behaviour, Actors elevate the process object to first class status.

Smalltalk, extended with Actors, provides facilities such as interprocess communication, multi-level dynamic priorities and processing accountability. Simulation designers are able to create processes of differing priority which have specific processing requirements and which can communicate with each other in a straightforward and useful fashion. Further, the implementation of these features is transparent to the designer.

2.1 Example Actors

Each Actor typically only does one job. While, in general, the types of Actors required and the jobs they do are application specific, there are a number of generic Actors which may be found in many applications. In particular, administrator, worker, server and courier Actors are common. Their roles are implied by their names. Administrators are in charge of resource allocation, worker coordination and system maintenance. Workers do the fundamental computation required by the application. Servers are used to manage a particular shared system resource such as a storage device or communications link. Couriers are responsible for transferring data from one Actor to another whenever it is available. Beyond these basic types however, we feel that not enough experience has been gained to allow for generalizations regarding the functionality, role or general utility of generic Actors. This lack of valid generalizations means that designers will, by-and-large, create Actors which are specific to their application.

2.2 Actor Communication

An Actor's basic communication requirement is to be able to talk directly with other Actors. How this is to be done varies from Actor to Actor. Administrators need to delegate tasks to their workers but cannot afford to wait until the tasks have been completed. Servers must be able to respond to requests immediately, perhaps postponing the actual computation being requested until some time.

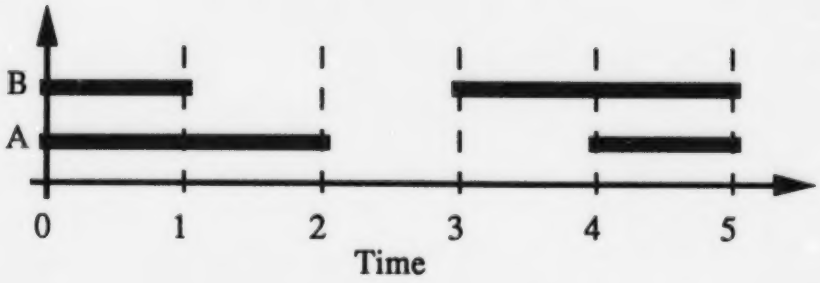
In the interest of creating a simple, easy-to-use system, a set of message passing operations would be preferable to a shared memory model of communication. The use of message passing allows for a higher level of abstraction and more transparency with respect to the mechanics of the protocol. Shared memory communication requires the programmer to wait and signal semaphores explicitly to synchronize the communicating parties. In large and complex applications this becomes very confusing. Two examples of message passing protocols are the rendezvous found in Ada and the scheme used in Harmony.

The rendezvous mechanism consists of a series of process entry points and points in time at which communication can take place. Process A, wishing to communicate with process B, calls for a rendezvous with B at some entry point (much like a remote procedure call). The rendezvous takes place when B signals that it is willing to accept a rendezvous at the specified entry point. During a rendezvous, uni- or bidirectional communication is possible. Process A is suspended from the time it calls for a rendezvous to the time B determines the rendezvous is complete. The rendezvous does not allow processes to defer ending the rendezvous (i.e., it is not directly possible for a process to keep a list of processes which are currently connected with it and choose to close the connection with particular ones). Rendezvous (entry) points are defined and fixed at compile-time and so cannot be added or removed dynamically. Further, the Ada language definition does not provide for a description of how tasks on different processors communicate or how tasks are allocated to processors. These factors vary from one real-time executive to another.

Harmony uses blocking sends and receives in conjunction with asynchronous replies to pass messages. When process A sends a message to process B, it will (block) until B replies. If B is ready to receive a message from others then it does a blocking receive (i.e., suspends until a message is sent to it). When a message arrives, B extracts the data (both sends and replies contain data) and processes it. B can reply to A at any time after it receives the message. When A receives the reply from B it unblocks and processes any data returned in the reply (see Figure 2.1). This model does not prohibit the implementation of a selective receive. Because it is not a restrictive predefined language construct, the code for determining if a particular process is currently willing to receive a particular message can be arbitrarily complex. The Harmony task/processor allocation is dynamic in that tasks can be created on any processor and can migrate from processor to processor during their lifetime.

The Harmony message passing protocol has a number of advantages which combine to give a wide range of possibilities within the same basic protocol.

- The simplicity of the primitive operation set affords those creating new operations more flexibility (Ada has a large and complicated primitive set).



0 : Both processes A and B are executing.
1 : B does a receive (i.e., waits for a message).
2 : A sends a message to B and waits for a reply (blocking send).
3 : B receives A's message and begins processing it.
4 : B replies to A who resumes processing.
5 : B finishes processing A's message.

FIGURE 2.1: TIME LINE OF INTERPROCESS COMMUNICATIONS IN HARMONY

- The receiving process' capability to reply to a message before finishing its processing allows fast communication over a many to one channel. This is ideal for fast, high capacity servers.
- It is possible to write generic servers whose services are dynamic (i.e., services can be added and removed at run-time). Adding services in an Ada system would require the addition of a task entry (rendezvous) point. This is impossible since the entry points are fixed at compile-time.
- Techniques for programming using the Harmony message passing protocol have developed over a number of years during which the Harmony mechanism has been used in large multiprocessor applications. Ada's implementation of the rendezvous mechanism is relatively new and is only now experiencing widespread use in large multiprocessor applications.
- Ada's tasking model has been found to be largely unusable for real-time applications because task scheduling is not predictable. This is shown by the existence of a multitude of different Ada real-time executives.
- Extracting the message passing mechanism from the rest of the system is more difficult for Ada than for Harmony because Ada's tasking is an integral part of the language and depends on other components of Ada. Consequently, we have elected to adopt the Harmony model for our implementation of Actors.

2.3 Software Development Features of Actors

The use of Actors in multiprocessing applications affords the designer and programmer several luxuries. The designer has a modelling tool which is highly intuitive and easy to understand. In many cases real world objects can be mapped on a one-to-one basis to Actors. Someone looking at the Actor model of a real world object can easily understand how it works, what its components are and how they fit together. Programmers do not have to worry about the details of process synchronization and communication as they do in other systems.

Actors fit into parallel and serial computation systems equally well. Applications are such that the code which runs on a system with ten processors will run on a system with one processor. Developers can experiment with performance by mixing parallel and serial computations. Actors are very modular thus Actor-based applications are portable and reusable. For example, two input/output (I/O) servers which respond to the same messages can be interchanged at will even if they are associated with entirely different kinds of devices.

2.4 Implementation of Actors in Smalltalk

This section discusses as unique implementation of Actors in Smalltalk done at Defence Research Establishment Ottawa (DREO) which drew on independent work carried out at Carleton University and the MIT Artificial Intelligence Lab. The aim of this work was to produce a system which could be used to support an experimental real-time system simulation environment.

Actors, as they are referred to here, are fully integrated members of Smalltalk. Actor message sends are the same as Smalltalk sends both in appearance and basic behaviour (i.e., they look the same and execution of the sending method is affected in the same way). Most importantly, the code required to implement Actors is simple, clear and concise.

An Actor is a Smalltalk object with an associated process. Normally, whenever a message is sent to an object, the currently executing process is responsible for the execution of the related method. With Actors, the process associated with the Actor (object) receiving the message is responsible for execution of the method. In this way the objects are autonomous; they are responsible for their own computational requirements.

2.4.1 Classes required for Actors

The Actor concept is implemented using the class Actor and two support classes, ActorMessage and Encapsulator. These classes form the essential definition of Actors. An ActorMessage is the container for data being passed from Actor to Actor while an Encapsulator is the Actor-to-Actor interface.

2.4.1.1 Actor

Instances of class Actor have the following instance variables.

- task

  The task is an instance of class Process and is the part of an Actor which is responsible for the computational power required to complete requested operations. The priority of a process (and thus an Actor) can be changed dynamically. Blocking (on sends and receives) is implemented by suspending and resuming the task; no semaphores are used.

- state

  The state instance variable reflects what the Actor as a whole is currently doing. Actors may be running, suspended, awaiting a message or awaiting a reply. If an Actor is running, it is not necessarily the currently running process (Smalltalk uses non-preemptive uniprocessor multitasking), but it is available to be run by the processor. A suspended Actor can only run again if some other Actor resumes it. Awaiting a message or reply is like being suspended, except that the Actor will run again upon receipt of the desired message; it cannot be resumed.

- mailbox

  When one Actor sends a message to another the message arrives in the destination Actor's mailbox which is either a first-in, first-out (FIFO) or priority queue. All messages received by an Actor go through its mailbox.

- encapsulator

  An Actor's encapsulator is an instance of class Encapsulator which is described below.

2.4.1.2 ActorMessage

ActorMessages are the containers for data which is to be sent from one Actor to another. They contain the source, destination and type of a message as well as the data being transferred.

2.4.1.3 Encapsulator

The fundamental concept of an Encapsulator is not new, having been suggested by (Ref. [8]). Encapsulators are used to unobtrusively (i.e., transparently) intercept all messages sent to a particular object. For example, it is possible to take an arbitrary object say, a String S, and encapsulate it so that every time the third character of S is accessed a bell is rung.

This behaviour can be built into the class String, but modifying the class is not the correct approach. The class String has over 2000 instances in a typical image and this modification would affect all of them. Every access to any String's third element would cause a bell to ring. In addition, adding monitoring code to a method reduces its readability and obscures its true functionality.

It is more useful and convenient to be able to encapsulate a particular object without modifying any of that object's code. This makes it possible to create, and switch between many different kinds of Encapsulator, some functioning as monitors and others as message transformers. Switching the Encapsulator for an object requires no code modification, rather a new Encapsulator is created and associated with the object.

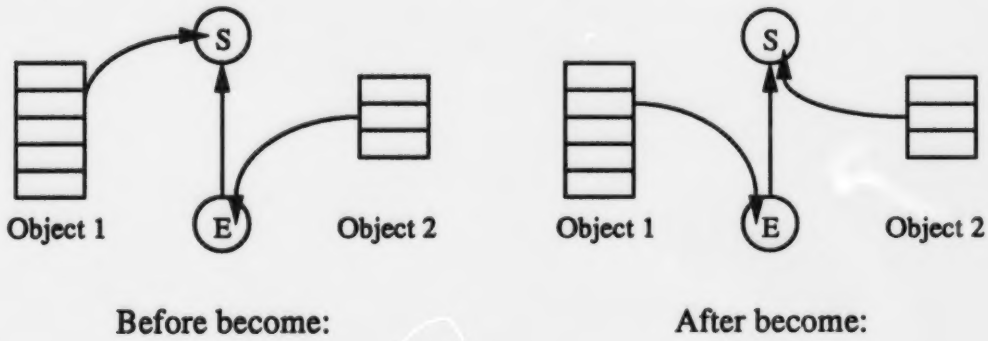


FIGURE 2.2: OBJECT POINTERS BEFORE AND AFTER A TWO WAY become: OPERATION

The encapsulation of a String, S, is effected by carrying out the following steps:

1. Create a new instance, E, of the appropriate Encapsulator subclass.

2. Associate E with S.

3. Swap all pointers to S for E and vice versa using the become: operation (see Figure 2.2).

Notice in Figure 2.2 that, before the become:, if Object 1 accessed its second instance variable it would get the String S while after the become: it would get E (vice versa for Object 2). The switch is transparent since all code which accesses instVar 2 thinking that it contains S will get E.

Step 3 is very dangerous at best and impossible at worst (some Smalltalk implementations do not provide two way become: operations). Since become: is used to automatically reassign to E all pointers which point to S, it is not necessary to use become: if, at the time of the encapsulation of S, all of the objects which point to S are known and their pointers can be manually changed to point to E.

Once E is in place, all messages which were originally directed to S will go to E, thus Encapsulators (E) must appear to be (i.e., behave like) the encapsulated object (S). E can now monitor S's message traffic and perform interception operations on incoming messages (e.g., allow some messages to pass through to S while holding others back). In this way E can modify the apparent behaviour S without modifying S.

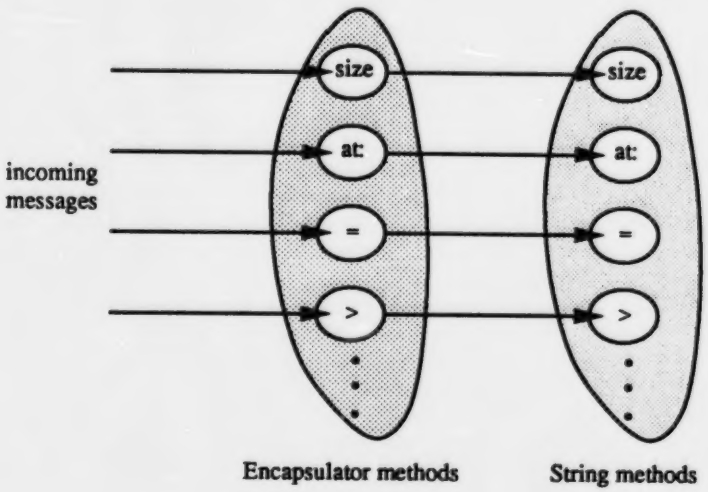


FIGURE 2.3: SIMPLE MESSAGE MONITORING TECHNIQUE

In the absence of some sort of generalized message trapping scheme, Encapsulator has to implement a method corresponding to each of the methods in the protocol for String (see Figure 2.3). These methods perform the desired interception operation for that message (e.g., messages can be forwarded, logged, tagged, stopped, ...). In this way, Encapsulator is a shadow of String. A different Encapsulator class is required for each class whose instances are to be encapsulated. The result is a shadow class hierarchy. There are far too many classes of objects to encapsulate for this approach to be feasible. Further, adding a method to a class requires the addition of a interception method to the corresponding Encapsulator class. Fortunately, the doesNotUnderstand: mechanism offers a solution to this problem. A closer look at the Smalltalk interpreter shows why.

When the Smalltalk interpreter encounters a message send, it does a method lookup on the message selector. A method lookup is done by climbing the class hierarchy from the receiver (the destination of the message) up to Object looking for the desired method. If a method is found then it is executed and a value returned. If a method is not found then the lookup fails and the message doesNotUnderstand: with the original message as the argument is sent to the original receiver.

By defining Encapsulators such that they do not inherit methods from any other class (including Object) and implement only the doesNotUnderstand: method, all incoming messages will be trapped by Encapsulator doesNotUnderstand: (see Figure 2.4). Removal of the inherited methods from Encapsulator can be done by defining Encapsulator as a subclass of nil or by removing all superclasses from the method search path. Which method is used depends on the particular implementation of Smalltalk being used. Consider the following example which illustrates how this mechanism is used to trap all messages to an object.

Suppose B is some object whose contents instance variable contains a String S. After encapsulating S with an Encapsulator E, B's contents will be E. If B wants to know the size of its contents then it executes the code, contents size, thus sending the size message to E. E is not able to respond to size (because it does not implement or inherit it) so the method lookup fails and a doesNotUnderstand: message is sent to E; the message is trapped. The doesNotUnderstand: method for E performs the desired interception operation on the message (it may or may not pass the message on to S) and then returns. This mechanism relies on the fact that the Encapsulator will not understand size (otherwise the doesNotUnderstand: message would not be sent). Note that only the inheritance chain for Encapsulator class needs to be broken; the chain for the Encapsulator metaclass can remain intact.

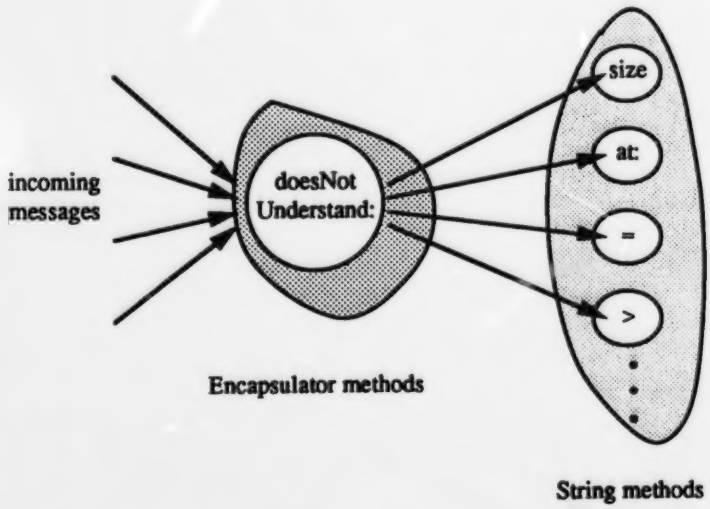


FIGURE 2.4: MONITORING MESSAGE TRAFFIC USING doesNotUnderstand:

The two interception methods discussed can be mixed thus excluding some messages from being caught by the DoesNotUnderstand: mechanism. If the message being received is implemented by the Encapsulator then doesNotUnderstand: is by-passed.

A variety of very sophisticated monitors can be implemented in a clean and transparent way due to the increased functionality provided by Encapsulators. For the remainder of this report the statement that some object is an Actor implies that the object is an Encapsulator encapsulating an instance of Actor.

2.4.2 How Actors Pass Messages

Encapsulators are used to make the Actor message sends look the same as Smalltalk message sends. The following two fragments of Smalltalk code appear the same and will accomplish the same computation but one sends a Smalltalk message to an object and the other sends an ActorMessage to an Actor.

```
someObject someMessageWith: arg1 with: arg1          (1)
someActor someMessageWith: arg1 with: arg2           (2)
```

Segment 1 is a Smalltalk message send because the receiver is an object while segment 2 is an Actor message send because the receiver is an Actor.

Encapsulators are used to transform what starts as a Smalltalk message send into an Actor message send. The doesNotUnderstand: method for an Actor's encapsulator changes the Smalltalk message into an Actor message and sends it to the intended receiver using a Harmony-like message passing protocol as described in section 2.2. Figure 2.5 below gives the pseudo-code for this doesNotUnderstand: method.

```
doesNotUnderstand: aMessage
    "Translate the Smalltalk message aMessage into an Actor
    message and send it to the intended receiver (the receiver
    of this message). Answer the result of the reply."

    build an Actor message using the information in aMessage
    send the message using the Harmony mechanism
    wait for a reply
    extract and return the result from the reply
```

FIGURE 2.5: PSEUDO-CODE FOR AN ACTOR ENCAPSULATOR'S doesNotUnderstand: METHOD

Multiprocessing coordination of Actors is done by suspending and resuming their associated processes rather than using semaphores. When an Actor blocks (either for a send or reply), its process is suspended and when the message it is waiting for arrives its process is resumed.

2.4.3 Monitoring

Encapsulators can be used to create powerful monitoring and debugging tools. Tools for both inter-Actor (Actor message) and intra-Actor (Smalltalk message) communication provide the user with a multi-level view of the environment. Actor message monitoring provides information regarding resource availability and processor usage. The ability to monitor and debug at all levels of interaction is an important feature of a comprehensive software development environment.

Implementations of Actor message monitors and debuggers was based on the standard Smalltalk inspector and code debugger. Significant changes to the code debugger were made so that it can recognize that an Actor send is about to take place and allow the user to debug (i.e., step through) the Actor message send. The method of executing primitives from the debugger and accessing instance variables was also changed. The result is an Actor system which is fully integrated with the Smalltalk environment.

3.0 ACTOR-BASED SIMULATION

Actors are well suited to modelling real world objects. It is much more intuitive to describe the behaviour of physical objects in terms of Actors and objects rather than in a series of equations or scripts. Objects can be modelled at varying levels of detail since Actors interact with each other in a clean, well defined way. The engine of a car, for example, can be described in terms of the results of its operation (e.g., fuel intake produces horsepower and torque) (see Figure 3.1), the fuel intake and exhaust emissions or the interactions between the engine components (e.g., pistons turn camshaft which opens valves etc.) (see Figure 3.2). The objects simulated can be modelled using anything from blanket approximations to detailed descriptions of their components. Models of differing levels of detail can be freely mixed within the same simulation. A detailed representation of an engine's mechanical components can co-exist and interact with a general model of the engine's electrical system. Of course the level of detail of a particular model can be changed without requiring changes to the other components in the system. Users need only use the amount of detail required by their own application.

There are no clear guidelines as to what should or should not be modelled as an Actor (i.e., one man's Actor is another's object). Because Actors are an extension of objects, both can exist in an Actor-based simulation. Actors provide the simulation designer with a transparent communication and synchronization mechanism for the components of the simulation. Modelling is quicker and more intuitive because the

focus is on what the individual objects do and how they behave, not on how to synchronize and coordinate the system as a whole. Actors are automatically coordinated when messages are passed between them. By introducing the notion of time Actors can also be synchronized. Because programmers do not have to make special efforts to facilitate communication, simulations are not clogged and confused with code not relevant to the model's behaviour.

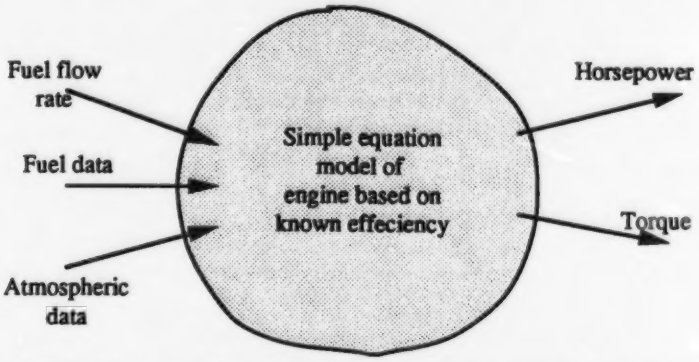


FIGURE 3.1: SIMPLE MODEL OF A REAL WORLD OBJECT

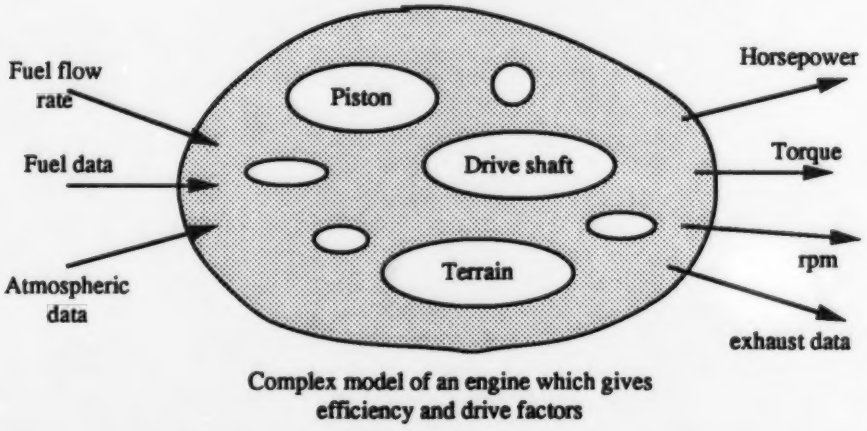


FIGURE 3.2: COMPLEX MODEL OF A REAL WORLD OBJECT

3.1 Time, Events and Actors

Simulation of real-time systems requires simulation of a real-time clock. In clock driven simulations, advancing the clock causes events to occur. In event driven simulations, the occurrence of events causes time to progress. Actor-based simulation combines these properties. Each Actor has a

clock which keeps its time. All Actors who have the same clock are synchronized, while Actors with different clocks can have different time. A multitasking uniprocessor system is a good example of the Actor/clock relationship. The processor itself is the clock while the tasks running on the it are the Actors. All the tasks have the same time (all are on the same processor and thus have the same clock) and the processor's time is advanced when the tasks run. If no tasks are running then the processor is stopped and time does not advance. If the system has multiple processors then there are multiple clocks and interprocessor communication requires clock synchronization (communications channel arbitration).

In the interest of maximum concurrency, Actors should run freely until forced to synchronize. Synchronization is required when Actors depend on one another (i.e., when messages are passed). Messages must arrive in an Actor's mailbox in the same order as they would in the real system. The important part of this constraint is that an Actor should not receive a message until after the time when it was sent. (Note: A message is not 'received' until the destination Actor takes it out of its mailbox and processes it.) For example, if an Actor at time 154 sends a message to another Actor who is at time 127, the destination Actor should not receive that message until after it has reached time 154. Since Actors use blocking sends, the sending Actor must wait at least until the receiving Actor reaches time 154. The two Actors are therefore synchronized.

To maintain this temporal consistency, the way a clock's time is advanced must be regulated. Many systems use TimeWarp (Ref. [7]) technology to maintain the simulation clock. Although developed independently by the author, the system used here is quite similar to the timeLock technique (Ref. [2]). The benefits and drawbacks of TimeWarp versus timeLock have not been fully investigated at this point and may be the subject of future investigation.

TimeLock dictates that when an Actor receives a message it advances its clock's time to be the greater of its current time and the message's send time plus some transmission delay. A clock's time can never go backwards. A message, M, should not be received by Actor A until it is clear that no messages with earlier send times can arrive in A's mailbox. A message with an earlier send time can be generated by an Actor whose time is earlier than M's. If such an Actor exists then A cannot receive M because it may be sent a message before M. If, for example, there are no Actors with time less than M's send time then A can proceed to receive M.

If this principle is ignored then the following problem arises. Suppose two messages, M1 and M2 are sent to A, and that A receives M1 first. M1's send time is 321 and M2's send time is 234 (note that M2 was sent before M1). Suppose both messages are requests for some resource. If A satisfies M1 with the last quantity of the resource then request M2 cannot be satisfied. In the real system, M2 would have been received first and it would have been request M1 which could not be satisfied. Subtle behavioural inconsistencies of this sort have serious effects on the validity of simulation results.

3.2 The Clock Hierarchy

A single level of time dependency is good for simple simulations but breaks down when the model being simulated becomes complex. Consider a multitasking multiprocessor system which allows interprocessor communication. To simulate this and maintain time consistency across the system an additional level of time dependency, system time, must be added to the simulation. Task time depends on processor time which in turn depends on system time. A hierarchy of clocks is required. Such a hierarchy must be flexible enough to allow Actors and clocks to depend on the same clock at the same time.

3.3 Implementation

Actor-based simulation is implemented using the Actor environment created in Smalltalk in addition to five classes specific to simulations. Original work on some facets of this implementation was done previously at the DREO (Ref. [1]) and supplied a specification of the basic functional requirements of the simulation system. Even though most of the original code and methodology was eventually removed, the original system provided a prototype from which to evolve.

The new implementation has many advantages over the old. Most importantly, the new simulation system is powerful, small, easy to understand and simple to use. As described below, only five classes were required and three of these are extremely simple.

3.3.1 Required Classes

The additional classes required are: SimulationActor, Clock, SimulationObject, SimulationSystem and SimulationMessage.

3.3.1.1 SimulationActor

SimulationActor is a subclass of Actor. Minor changes to the functionality of Actors were required to handle time synchronization for message sends and to manipulate the clock.

Messages are time-stamped with both the send time (when the message was sent) in terms of the sender's clock and the received time (when the message was received) in terms of the receiver's clock.

The predicate used to check if a message is receivable or not now checks the send times of the messages in the Actor's mailbox against the Actor's time (its clock's value). A message cannot be received until the destination Actor's time is greater than the message's send time.

If an Actor finds no receivable messages in its mailbox it must try to advance its clock's time to the send time of the first message in the mailbox. The clock's time will be advanced if it is impossible for some other Actor to send a message which will have an earlier send time.

When a SimulationActor sends a message its clock is advanced to the send time of the reply which is returned plus some transmission delay.

3.3.1.2 Clock

Clocks, a subclass of Object, consist of a time value and a mutual exclusion semaphore. Time is advanced by first obtaining the semaphore and then modifying the time. A clock can be set to a specific value or incremented by some delta value. A clock's value should never decrease during normal running of a simulation as this would cause inconsistencies in the time-stamping of messages received.

3.3.1.3 SimulationObject

SimulationObject, a subclass of Clock, is one of the fundamental elements of a simulation. All major components in a simulation which are not SimulationActors are SimulationObjects. A SimulationObject is a mechanism for grouping SimulationActors and other SimulationObjects together to form logical units. Consider, for instance, a processor in a computer system as an example of a SimulationObject. Tasks (SimulationActors) are associated with the processor which maintains time for them. The tasks within a processor interact with each other and thus cause the progression of time. The functionality of the processor (SimulationObject) allows it and other objects to be grouped together to form a simulated computer system (a SimulationSystem). The new system can be combined with other systems, processors and tasks (SimulationSystems, SimulationObjects and SimulationActors) to form yet another system, and so forth.

3.3.1.4 SimulationSystem

The main purpose of SimulationSystem, a subclass of SimulationObject, is to support the clock hierarchy concept.

SimulationObjects are grouped together in a system, say S, and S coordinates all the objects contained in it. The system may itself be an element in another system, say P, on the next higher level in the clock hierarchy. When S tries to change its own time it must first coordinate with P just as the objects in S had to coordinate with S.

Systems on different branches of the hierarchy run independently in the absence of message sends since only elements which send and receive messages need to be coordinated. Note that only the objects involved in a message transaction (and those above them in the hierarchy) are affected by a message send, the other elements in the simulation will continue to run uninterrupted.

3.3.1.5 SimulationMessage

SimulationMessage, a subclass of ActorMessage, is the basic unit of data in the simulation. It has additional fields and protocol for message send and receive time-stamps.

3.3.2 SimulationMonitor

A normal Smalltalk debugger presents the user with what is essentially a message send stack. The contexts appearing in the context view are those which have been created as a direct result of Smalltalk message sends. This allows the user to backtrack, see how and why particular pieces of code are being run and change the values being passed. In an Actor-based system it is more interesting to see the message traffic between Actors than between objects.

A SimulationMonitor is to Actor message sends what the debugger is to Smalltalk message sends. Messages are maintained on a stack which can be inspected allowing the user to change the data being passed. Using a monitor in conjunction witf the Smalltalk debugger gives the user two levels of control. The monitor is used to debug inter-Actor communication while the debugger is used to debug individual Actors.

SimulationMonitors can be opened on individual SimulationActors, SimulationObjects or SimulationSystems. When monitoring objects or systems the user can select particular components to monitor. In addition, monitors provide multi-level message tracing. In verbose mode, all message sends, receives, replies and reply receipts are trapped by the monitor, while in normal mode only the message receives and replies are trapped. Of course, tracing can be turned off altogether.

The user can also set breakpoints on all messages, particular messages for all Actors, or particular messages for particular Actors. When a breakpoint is encountered execution halts in the code for the destination Actor just prior to the receipt of the message. At this point, messages on the message stack may be inspected and modified and execution continued. When an Actor is halted, the user may inspect or debug the associated task using the available Smalltalk tools.

4.0 HARMONY SIMULATION/EMULATION

Harmony is the real-time operating system whose message passing protocol was adopted for our implementation of Actors. Because of this, emulating Harmony using the simulation system is straightforward. The basic requirement is the ability to write tasks for a Harmony system and run simulations using these tasks. To do this a number of features of a Harmony system are required.

A Harmony application is a collection of tasks, some of which are system/administration oriented while others are application specific. These tasks are downloaded to one or more processors in one or more systems and started running. Individual tasks can register with a directory thus giving themselves a global name. In general, tasks are created and killed, messages are passed and the system does some work. Remembering that Actors model the basic feature of Harmony, its message passing, how is Actor-based simulation best used to emulate these parts and thus a whole Harmony system?

4.1 An Implementation of Harmony Emulation

The implementation of the Harmony emulation is comprised mainly of the classes HarmonyTask, HarmonyDirectory, HarmonyProcessor and HarmonySystem.

4.1.1 HarmonyTask

In describing Actors, the task as an Actor analogy has been used several times. Here the reverse is true. The class HarmonyTask is a subclass of SimulationActor. HarmonyTasks have additional capabilities which allow them to spawn and manage child tasks. The parent of a task is initially its creator, but it can be changed. Killing a task kills all of its children. Programmers using the Harmony emulation create subclasses of HarmonyTask to fulfill their specific requirements.

4.1.2 HarmonyDirectory

A HarmonyDirectory is a task (HarmonyTask) which maintains a global name space. Tasks register with the directory (typically there is only one directory per system) under some predetermined name when the system is started. Using this mechanism, tasks can reference each other using meaningful symbolic names rather than dynamically determined numerical ids. The dictionary is keyed on task names and points to the actual tasks. Operations for registration and deregistration of tasks are supplied. The directory task is globally known so new tasks know where to find it.

4.1.3 HarmonyProcessor

The class HarmonyProcessor is a subclass of SimulationObject. The processor is responsible for supplying the tasks running on it with time information and coordinating itself and its tasks with the other processors in the system.

4.1.4 HarmonySystem

A HarmonySystem is a type of SimulationSystem whose components are HarmonyProcessors. The use of a HarmonySystem to group processors and their related tasks serves several purposes. It allows intertask communication across processors because of the coordination properties of SimulationObjects in a SimulationSystem. HarmonySystems also provide a convenient grouping mechanism which is used in system-wide operations (e.g., booting, shutting down, etc.).

4.2 Results

Harmony emulation using Actor-based simulation works quite well. In addition to simple emulation it is possible to implement Harmony monitors which allow the user to interactively monitor and debug message traffic between Harmony tasks. Harmony application and system tasks were created and run, successfully completing their assigned jobs.

5.0 CONCLUSIONS

The simulation system described here has properties of both clock-based and event-based systems. Both can be implemented separately using Actor-based simulation. Simulation designers can also mix paradigms within the Actor framework, combining the features of both to solve complex problems more intuitively.

This work has resulted in a simulation system which has interactive monitoring and debugging facilities at the level of component (Actor) interaction (message passing) and component functionality (Actor code). As demonstrated by the Harmony emulation, Actor-based simulation is a powerful modelling tool which can be used to simply and quickly create models of real world systems. Objects are modelled in an intuitive fashion by breaking them down into parts at the desired level of detail. The detail or accuracy of the developed simulation can be changed by varying the number of parts and inter-part dependencies used in the model (e.g., to more accurately model an engine, model the pistons and the gas exchanges rather than just the crankshaft). Changes to a simulation are easily implemented since the components encapsulate functionality and support well-defined interfaces with other components.

Simulations produced using this system are simple, easy to understand and use, and very modular.

6.0 REFERENCES

[1] Barry, B., "Object-Oriented Simulation of Electronic Warfare Systems", Proceeding of the 1987 Summer Computer Simulation Conference, p.759-764, Montreal, Canada, July 1987.

[2] Bézivin, J. and Imbert, H., "Adapting a Simulation Language to a Distributed Environment", Proceedings of the 3rd International Conference on Distributed Computing Systems, p.596-605, Miami, Florida, October 1982.

[3] Digitalk Inc. (1986) Smalltalk/V Tutorial and Programming Handbook, Digitalk Inc., Los Angeles, CA. Smalltalk/V is a registered trademark of Digitalk Inc.

[4] Gentleman, M., "Using the Harmony Operating System", ERB-966, NRCC No. 24685, National Research Council of Canada, Ottawa, Ontario, 1985.

[5] Goldberg, A. and Robson, D., "Smalltalk-80: The Language and Its Implementation", Addison-Wesley Publishing Company, Don Mills, Ontario, 1983.

[6] Hewitt, C., Bishop, P. and Steiger, R., "A Universal Modular Actor Formalism for Artificial Intelligence", IJCAI-73, Stanford, CA, 1973.

[7] Jefferson, D.R. and Sowizral, H., "Fast Concurrent Simulation Using the Time Warp Mechanism", Proceedings of the Conference on Distributed Simulation 1985, p.63-69, San Diego, January 1985.

[8] Pascoe, G., "Encapsulators: A New Software Paradigm in Smalltalk-80", Proceedings of OOPSLA '86, ACM SIGPLAN Notices, volume 21, number 11, p.341-346, 1986.

[9] Smalltalk-80 is a registered trademark of XEROX, Inc.

7.0 ACKNOWLEDGEMENT

The author would like to acknowledge the colleagues who offered their time and advice, including Brian Barry for his initial work, time spent listing and thinking about the many problems encountered and for editing this report, and the whole AMEP team for providing various system components and their opinions. Special thanks are due to Guy Farley for his expert translation.

UNCLASSIFIED

SECURITY CLASSIFICATION OF FORM
(highest classification of Title, Abstract, Keywords)

DOCUMENT CONTROL DATA

(Security classification of title, body of abstract and indexing annotation must be entered when the overall document is classified)

1. ORIGINATOR (the name and address of the organization preparing the document. Organizations for whom the document was prepared, e.g. Establishment sponsoring a contractor's report, or tasking agency, are entered in section 8.)

NATIONAL DEFENCE
DEFENCE RESEARCH ESTABLISHMENT OTTAWA
SHIRLEY BAY, OTTAWA, ONTARIO, K1A 0Z4 CANADA

2. SECURITY CLASSIFICATION (overall security classification of the document including special warning terms if applicable)

UNCLASSIFIED

3. TITLE (the complete document title as indicated on the title page. Its classification should be indicated by the appropriate abbreviation (S,C,R or U) in parentheses after the title.)

A SIMULATION SYSTEM BASED ON THE ACTOR PARADIGM (U)

4. AUTHORS (Last name, first name, middle initial. If military, show rank, e.g. Doe, Maj. John E.)

MCAFFER, J.

5. DATE OF PUBLICATION (month and year of publication of document)

FEBRUARY 1989

6a. NO. OF PAGES (total containing information. Include Annexes, Appendices, etc.)

24

6b. NO. OF REFS (total cited in document)

9

6. DESCRIPTIVE NOTES (the category of the document, e.g. technical report, technical note or memorandum. If appropriate, enter the type of report, e.g. interim, progress, summary, annual or final. Give the inclusive dates when a specific reporting period is covered.)

DREO TECHNICAL NOTE 89-4

8. SPONSORING ACTIVITY (the name of the department project office or laboratory sponsoring the research and development. Include the address.)

NATIONAL DEFENCE
DEFENCE RESEARCH ESTABLISHMENT OTTAWA
SHIRLEY BAY, OTTAWA, ONTARIO K1A 0Z4 CANADA

9a. PROJECT OR GRANT NO. (if appropriate, the applicable research and development project or grant number under which the document was written. Please specify whether project or grant)

011LB14

9b. CONTRACT NO. (if appropriate, the applicable number under which the document was written)

10a. ORIGINATOR'S DOCUMENT NUMBER (the official document number by which the document is identified by the originating activity. This number must be unique to this document.)

10b. OTHER DOCUMENT NOS. (Any other numbers which may be assigned this document either by the originator or by the sponsor)

11. DOCUMENT AVAILABILITY (any limitations on further dissemination of the document, other than those imposed by security classification)

(X) Unlimited distribution
( ) Distribution limited to defence departments and defence contractors; further distribution only as approved
( ) Distribution limited to defence departments and Canadian defence contractors; further distribution only as approved
( ) Distribution limited to government departments and agencies; further distribution only as approved
( ) Distribution limited to defence departments; further distribution only as approved
( ) Other (please specify):

12. DOCUMENT ANNOUNCEMENT (any limitation to the bibliographic announcement of this document. This will normally correspond to the Document Availability (11). However, where further distribution (beyond the audience specified in 11) is possible, a wider announcement audience may be selected.)

UNCLASSIFIED

SECURITY CLASSIFICATION OF FORM

UNCLASSIFIED

SECURITY CLASSIFICATION OF FORM

13. ABSTRACT (a brief and factual summary of the document. It may also appear elsewhere in the body of the document itself. It is highly desirable that the abstract of classified documents be unclassified. Each paragraph of the abstract shall begin with an indication of the security classification of the information in the paragraph (unless the document itself is unclassified) represented as (S), (C), (R), or (U). It is not necessary to include here abstracts in both offical languages unless the text is bilingual).

(U) The main goal of this work is to produce a sophisticated object-oriented design tool for simulating real-time applications. The basis of this tool is Actor-based simulation, a powerful mechanism for modelling real world objects simply, quickly and intuitively. It combines features of event-based and clock-based simulation and allows designers to use both in a common framework. Actors can be used to represent everything from the most basic to the most complex components. Complex objects are constructed by modelling their subcomponents and interactions. The Actor paradigm provides a good model of concurrency and facilitates the development of comprehensive monitoring and debugging tools.

14. KEYWORDS, DESCRIPTORS or IDENTIFIERS (technically meaningful terms or short phrases that characterize a document and could be helpful in cataloguing the document. They should be selected so that no security classification is required. Identifiers, such as equipment model designation, trade name, military project code name, geographic location may also be included. If possible keywords should be selected from a published thesaurus, e.g. Thesaurus of Engineering and Scientific Terms (TEST) and that thesaurus-identified. If it is not possible to select indexing terms which are Unclassified, the classification of each should be indicated as with the title.)

Object-oriented simulation
Actor-based programming
Simulation
Object-oriented programming
Smalltalk

UNCLASSIFIED

SECURITY CLASSIFICATION OF FORM